Verse: An EDSL for cryptographic primitives

Abhishek Dang
Department of Computer Science and Engineering
IIT Kanpur
Kanpur, Uttar Pradesh, India
ahdang@cse.iitk.ac.in

Piyush P Kurur
Department of Computer Science and Engineering
IIT Kanpur
Kanpur, Uttar Pradesh, India
ppk@cse.iitk.ac.in

ABSTRACT

Cryptographic primitives need high-speed implementations that are also resistant to side channel attacks. The absolute control over instructions and registers that such implementations demand makes assembly language programming a necessity. In this article, we describe Verse, a *typed low-level* language *embedded* in Coq designed specifically to generate assembly language programs for cryptographic primitives. Despite being a low-level language, the programming experience is markedly high-level:

- The type system of Verse is rich enough to even prevent errors in array indexing and endian conversion.
- Being embedded in Coq, we have at our disposal Gallina, the underlying functional programming language, as a macro assembler for code generation, and Ltac, the tactic language, as an automation tool for proof obligations inherent to our type system.

We also provide a generic framework to formulate semantic aspects of Verse. This framework has value beyond providing an interpretation of Verse in Coq. We demonstrate this versatility by using it to localise uninitialised/clobbered variable use, and arithmetic overflows.

CCS CONCEPTS

• **Software and its engineering** → **Domain specific languages**;
• **Security and privacy** → *Logic and verification.*

KEYWORDS

Coq, EDSL, Cryptography, assembly language

ACM Reference Format:
Abhishek Dang and Piyush P Kurur. 2018. Verse: An EDSL for cryptographic primitives. In *The 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18), September 3–5, 2018, Frankfurt am Main, Germany*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3236950.3236971

1 INTRODUCTION

High-level languages with their ever improving compilers have made assembly language programming mostly irrelevant. However, implementing cryptographic primitives is one application domain where there is still some justification for using an assembly language. While performance is an obvious motivation for it, programming in assembly language is often necessary to guard against various *side channel attacks* - attacks that make use of information that is leaked inadvertently by implementations. Minimising side channel leaks requires precise control over the machine instructions that are executed when performing a cryptographic operation. The output of modern optimising compilers is too unpredictable for such a setting. For example, in its dead code elimination phase, a compiler might eliminate the code that wipes a secret password from a memory location; it might short circuit certain branching instructions based on cryptographic secrets, thus leaking sensitive information through timing data.

Programming in an assembly language, due to its inherent low-level nature, is tedious and error prone. Moreover, code written for a particular architecture cannot easily be ported over to a different architecture. This incurs a significant maintenance overhead. In this article, we describe Verse, an *embedded domain specific language* (EDSL) in Coq for generating low-level assembly language implementations of cryptographic primitives, which attempts to solve some of these problems.

Verse is designed to address two seemingly irreconcilable goals - provide the safety and portability of a high-level language, and yet be extremely close to the underlying machine so that cryptographic side channels can be controlled. At its core, Verse is a *typed* low-level language with an instruction set that is in one-to-one correspondence with the underlying machine. It gives a lot of control to the programmer, including, for example, the ability to control aspects like register allocation. Such precise control over sensitive data helps avoid many side channel attacks that are very hard to prevent in a typical high-level language. However, Verse also borrows a few features from high-level languages, prominent among which is a type system. Its type system is strong enough to prevent many common programming errors, including out of bound array indexing, at compile time (Section 3). A feature unique to Verse is its ability to track endianness of array variables. Not only does it prevent a lot of errors, it also adds to the portability: a verse program need not be rewritten just to make it work of an architecture with a different endianness (Section 3.4).

Cryptographic implementations need mostly arithmetic and bit wise operations. The instruction set of Verse is a common notation for these operations which makes it possible to share code fragments across supported architectures and thus regain some of the portability of high-level languages. Currently, Verse supports code generation for `X86_64` and portable C.

Finally, being embedded in a powerful language, Verse avoids the tedium that is often associated with programming in a low-level language by the use of Coq features like sectioning, modules, functions and tactics. In particular, we can use Coq functions as macros to generate repetitive code patterns.

2 A TOUR OF VERSE THROUGH EXAMPLES

Programming in Verse is carried out in two stages. In the first stage, the Verse programmer writes a generic program which can potentially be targeted at multiple architectures. This stage, while largely independent of the target architecture, needs to be careful about its instruction selection. Code generation in Verse is, mostly[1], a one-to-one translation of its instructions to those of the underlying architecture. Thus, for example, if the target architecture is X86, a program that uses the instruction `X ::= Y [+] Z` will fail during code generation because of the lack of support for this 3-operand instruction in X86. Beyond such considerations, Verse is similar to a typed low-level language like LLVM assembly. However, being an embedded coding environment, Verse code can be generated via macros written in Gallina, the functional language underlying Coq. Using macros Verse provides, or defining problem specific ones, makes programming relatively abstract.

In the second stage - code generation - the user needs to designate which program variables are parameters and which local variables are to be allocated into registers, and then give an explicit allocation into machine registers for the latter (at this time the target architecture is fixed). While the code generator takes care of parameter allocation according to the C calling convention and offset calculation for the local variables spilled on to the stack, the programmer cannot be completely oblivious of the target architecture. Calling conventions often reserve registers for parameter passing. The code generator checks for misuse of already allocated register and careless allocation can lead to frustrating failures during code generation. This explicit allocation, a minor irritant sometimes, gives total control over register use and is imperative to keep track of the locations of sensitive data.

We illustrate this two stage process by giving a sample program to implement a simplified[2] form of SHA2-like message scheduling where we take an array `w` of length 16 and perform the following update (index additions done modulo 16).

```
w[i] += w[i + 14] + w[i + 9] + w[i+1].
```

We begin by importing the top level Verse module.

```
Require Import Verse.
```

```
Module SHA2.
```

The standard idiom for writing a Verse program is to define a Coq **Section** which contains definitions of the program variables, Verse code, and other auxiliary information.

[1]The only exceptions are some additional byte swap instructions that might be added at the time of array indexing.

[2]The actual schedule uses logical functions σ_0 and σ_1 which we have skipped to simplify the example

```
Section SCHEDULE.
```

The message schedule of SHA512 and SHA256 involves the same message indices. In our simplified variant, the only difference between the two is that the former uses words of 64-bit whereas the latter uses words of 32-bit. By making the `word` a **Variable** of this section, our code effectively becomes polymorphic on the `word` type.

```
  Variable word : type direct.
```

The `type direct` in the above definition is the type of all direct types, i.e. types that fit into machine registers, which includes the word and multi-word types. See Section 3.1 for more details on types and kinds in Verse.

A SHA2 block is a 16 length array of this word type encoded in big endian.

```
  Definition SIZE := 16.
  Definition BLOCK := Array SIZE bigE word.
```

Generic variants of code are *parameterised* over the program variables which will eventually be instantiated from the architecture specific register set during code generation. In Verse the type `VariableT` is the universe of all possible program variable types.

```
  Variable progvar : VariableT.
```

What follows is the Verse program for message scheduling. First we "declare" the program variables, the variable `W`, `S`, and `T` followed by the actual message schedule.

```
  Variable W : progvar BLOCK.
  Variable S T : progvar word.

  Definition WordSchedule i (boundPf : i < SIZE) : code progvar.
    verse [ S ::== W[- i -];
            T ::== W[- (i + 14) mod SIZE -];
            S ::=+ T;
            T ::== W[- (i + 9) mod SIZE -];
            S ::=+ T;
            T ::== W[- (i + 1) mod SIZE -];
            S ::=+ T;
            MOVE S TO W[- i -]
          ].
  Defined.
```

Having defined the code segment for scheduling a single word in the message, we use the `foreach` function to generate an unrolled loop performing the `WordSchedule` for every index of `W`.

```
  Definition Schedule := foreach (indices W) WordSchedule.
```

This completes the Verse code for word scheduling. Before moving on to the code generation phase we digress and illustrate a powerful feature of Verse, namely *compile time* array bound checking. For example, all array indexing of the form `W[- x -]` in the code fragment `WordSchedule` and `Schedule` above are indeed within the bounds. We give a quick description on how this works out for the user when developing Verse programs using an interactive environment like proof general. The standard idiom in Verse for defining code segments is to introduce them through Ltac mode as shown in the definition of WordSchedule. Every array indexing `W[- x -]` generates the proof obligation `x < SIZE`. The `verse` "keyword" used above is in fact a tactic that tries to dispose of such obligations.

Out of bound array indices raise unsatisfiable proof obligations which, clearly, are impossible to dispose of. As a result the user is prevented from closing the above Definition, thus ensuring correctness of array indexing. To see this in action, let us drop `mod SIZE` in the expression `W[- (i + 1) mod SIZE -]` in the definition of `WordSchedule`. This results in the following contents in the *goals* buffer clearly showing an undisposed proof obligation `i + 1 < SIZE`.

```
1 subgoal, subgoal 1 (ID 154)

  word : type direct
  progvar : VariableT
  W : progvar BLOCK
  S, T : progvar word
  i : nat
  H : i < SIZE
  ============================
  i + 1 < SIZE
```

In addition, the tactic `verse` prints the following helpful hint in the *response buffer*

```
verse: unable to dispose of (i + 1 < SIZE)
possible array index out of bounds
```

Before we close the Coq section `SCHEDULE`, we need some additional definitions relevant to code generation.

```
    Definition parameters : Declaration := [Var W].
    Definition stack : Declaration := [].
    Definition registers : Declaration := [Var S ; Var T].
```

Code generators expect the parameters to be listed first, followed by the stack variables, and finally the register variables. The order within the lists needs to be consistent with the listing in the section.

```
  End SCHEDULE.
End SHA2.
```

2.1 Code generation

Generating a callable function out of Verse code requires us to give the register allocation for all the variables declared in the `SHA2.registers`. Allocation of `SHA2.parameters` and `SHA2.stack` is handled by the architecture specific frame management routines in the code generator. As a demonstration, we use portable C as the target architecture.

```
Require Import Verse.Arch.C.
Definition code64bit : Doc + {Compile.CompileError}.
  C.Compile.function "schedule64bit"
                     (SHA2.parameters Word64)
                     SHA2.stack
                     (SHA2.registers Word64).
  assignRegisters (- cr Word64 "S" , cr Word64 "T" -).
  statements (SHA2.Schedule Word64).
Defined.

Definition code32bit : Doc + {Compile.CompileError}.
  C.Compile.function "schedule32bit"
                     (SHA2.parameters Word32)
                     SHA2.stack
                     (SHA2.registers Word32).
  assignRegisters (- cr Word32 "S", cr Word32 "T" -).
  statements (SHA2.Schedule Word32).
Defined.
```

In case one wants to generate the code for a different architecture, one needs to use the appropriate code generator. Thus, a large part of the code can be used across architectures. The above example also demonstrates a different kind of code reuse. Both the 64-bit and 32-bit variants are generated from the same generic Verse program by instantiating the section variable `word`, which gives a kind of polymorphism, much like the template system of `C++`.

We generate the program text from the above code as our final step. Compilation of a Verse program may fail with a compiler error, say due to the use of an unsupported instruction, but if successful, will generate its output as a pretty printed object captured by the type `Doc`. The function `tryLayout` converts the compilation output to a string, which, in this case, is the corresponding C source code.

```
Compute (tryLayout code64bit).
```

The generated C code is given below:

```
#include <stdint.h>
void schedule64bit(uint64_t p0[16])
{
    /*    Local variables    */

    /*    Register variables    */
    register uint64_t rT;
    register uint64_t rS;


    rS =  Verse_from_be64(p0[0]);
    rT =  Verse_from_be64(p0[14]);
    rS += rT;
    rT =  Verse_from_be64(p0[9]);
    rS += rT;
    rT =  Verse_from_be64(p0[1]);
    rS += rT;
    p0[0] = Verse_to_be64( rS);

    /* similar lines skipped for conciseness */

    }
```

Notice that Verse automatically takes care of appropriate endian conversions without any intervention from the user.

2.2 Iterative functions

The last example demonstrated the use of Verse to implement a straight line program. Cryptographic primitives like hashes and ciphers process a stream of data which is chunked up into fixed size *blocks*, each of which is an array of a certain word type. For example, for `SHA512`, the *block* is a 16 length array of 64-bit words encoded in big endian. Verse gives a relatively high-level interface to write the block processing function for such primitives. Additional algorithms like padding strategy are not handled and need to be provided separately.

Recall that in the case of a straight line program, we defined a Verse program as a generic Gallina function that returns `code v` parameterised over the program variable. In the case of an iterative function, we define a program as a record of type `iterator blockTy v` instead.

```
Record iterator (blockTy : type memory)(v : VariableT)
  := { setup : code v;
         process : ∀ (block : v memory blockTy), code v;
         finalise : code v
       }.
```

In the above definition the `blockTy` denotes the block type used by the primitive (`Array` 16 `bigE Word64` for `SHA512`).

The block processing function is assumed to get its input, few blocks at a time, in a buffer. Between each of these buffer operations, it needs an internal state that is passed via parameters to the iterator function. The `setup` code initialises the function variables from the parameters before it starts processing the buffer, whereas the `finalise` code updates the state of the algorithm so that the next buffer can be processed. It is the `process` code fragment, that determines what needs to be done on a single block. The generated code takes care of looping over each block provided in the buffer and applying `process` on each of the them. Thus `process` needs to worry only about a single block.

2.3 Sample iterator

We illustrate iterator functions using the following computational task that models a highly *insecure* cipher: a block for the cipher is an array of four 64-bit words. The key also consists of four 64-bit words, and one needs to xor the key to the blocks. The last word in the key is to be treated as a counter and incremented after each block is processed. The code sample also illustrates Verse support for *array caching*, a coding pattern where an array whose entries are frequently used is held in registers to save memory access.

```
Module XORK.

  Definition SIZE := 4.
  Definition BLOCK := Array SIZE littleE Word64.
  Definition KEY := Array SIZE hostE Word64.

  Section XORKey.
    Variable progvar : VariableT.
    Arguments progvar [k] _.

    Variable Key : progvar KEY.
```

In this sample code, we do not use the array `Key` directly. Instead, we maintain a cache of it in the following variables.

```
    Variable K0 K1 K2 K3 : progvar Word64.
```

With this register cache in place, we first load the word `Key[- i - ]` in $\mathtt{K}_i$. While processing the blocks, we use the variable $\mathtt{K}_i$ in place of `Key[- i -]`. This variant is likely to be faster as we save on memory accessing needed to index elements of `Key`. However, unlike array elements `Key[- i -]`, we do not have the ability to refer to the register variables $\mathtt{K}_i$ uniformly. Without such indexing, helper Gallina functions like `foreach` would be useless and one would need to write the code by hand, which is clearly tedious. Verse exposes a set of helper functions to reduce such boilerplate.

```
    Definition KeyCache : VarIndex progvar SIZE Word64
      := varIndex [K0; K1; K2; K3]%vector.

    Definition keySetup := loadCache Key KeyCache.
```

The above definition makes `KeyCache` a function that maps array index `i` to the variable $\mathtt{K}_i$ and the code fragment `keySetup` loads the array into the register cache. We now give the block transformation code (which makes use of an additional temporary register).

```
    Variable Temp : progvar Word64.

    Definition xorIth (blk : progvar BLOCK) (i : nat) (_ : i < SIZE)
      : code progvar.
      verse [ Temp ::== blk[- i -];
                  Temp ::=^ KeyCache i _;
                  MOVE Temp TO blk[- i -]
                ].
    Defined.

    Definition blockTransform (blk : progvar BLOCK)
      := let incrementKey := [K3 ::=+ Ox "0000:0000:0000:0001"]
         in foreach (indices Key) (xorIth blk) ++ incrementKey.
```

The changes to the cached variable `KeyCache` need to be moved back to the array `Key` so that the cipher can xor further blocks. If all the cached variables were updated we could use the Verse library function `moveBackCache`. In this case however, it is more efficient to only move $\mathtt{K}_3$ since it is the only cached variable that got modified in the process.

```
    Definition keyFinalise : code progvar.
      verse [ MOVE K3 TO Key[- 3 -] ].
    Defined.
```

We package the setup, block processing, and finalisation routine into an iterator record, which will be used by the code generator.

```
    Definition Iterator : iterator BLOCK progvar :=
      {|
        setup := keySetup;
        process := blockTransform;
        finalise := keyFinalise
      |}.
```

As before, we need to declare the parameters, stack and register variables and, finally, compile using the `C.Compile.iterator` tactic.

```
    Definition parameters : Declaration := [Var Key].
    Definition stack : Declaration := [].
    Definition registers : Declaration
      := [Var K0; Var K1; Var K2; Var K3; Var Temp].

  End XORKey.

Definition code : Doc + {Compile.CompileError}.
  C.Compile.iterator XORK.BLOCK "xorblocks"
                              XORK.parameters
                              XORK.stack
                              XORK.registers.
  assignRegisters (- cr Word64 "k0"
                         , cr Word64 "k1"
                         , cr Word64 "k2"
                         , cr Word64 "k3"
                         , cr Word64 "temp"
                           -).
  statements XORK.Iterator.
Defined.

End XORK.
```

The skeleton of the resulting C function code is given below. Notice the two additional arguments - `blockPtr` and `counter` - that have been included in the function signature (on top of the declared parameter list) and the internal while loop which goes over the list of blocks in the buffer `blockPtr`.

```
void xorblocks( uint64_t ( *blockPtr)[4],
                uint64_t counter,
                uint64_t p0[4]
              )

{
    /* initialisation skipped */

    /*    Register variables    */
    register uint64_t rtemp;
    register uint64_t rk3;
    register uint64_t rk2;
    register uint64_t rk1;
    register uint64_t rk0;


    rk0 =  p0[0];
    rk1 =  p0[1];
    rk2 =  p0[2];
    rk3 =  p0[3];

    /* Iterating over the blocks */
    while(counter > 0)
    {
        /* skipped the body */
       ++blockPtr; --counter; /* move to next block */
    }
    p0[3] = rk3;
}
```

While the code above was written for exposition, an implementation of the Chacha20 cipher is available in our source code repository[3]

3 THE DESIGN OF VERSE

Verse is designed to be as close to the underlying machine as possible. The other consideration that went into its design was its use case - writing the inner loops of cryptographic primitives that are meant to be called by programs or libraries written in a high-level language. Functions written by Verse follow the C calling convention and hence Verse routines can be called by C or, for that matter, any high-level language like OCaml or Haskell that supports FFI calls to C. This focus allows for a rather simple design of the core language. Verse programs are, more or less, just lists of instructions and are designed to write only two kinds of functions (1) functions that are straight line programs and (2) functions that iterate over a sequence of blocks, as demonstrated in the previous section. These two classes of functions capture the essence of cryptographic primitives where the former takes care of fixed input primitives like elliptic curve signature schemes while the latter takes care of bulk primitives like cryptographic hashes, MAC's and ciphers.

The simple design of Verse has the following consequences. Apart from the implicit loop that is generated by the iterator, all loop-like constructions generated using `foreach` are unrolled into a list of instructions. This might seem like a limitation of Verse. However, genuine loops require support for conditional branching in assembly, which, if not carefully sanitised, can leak side channel information due to branch prediction logic in modern processors. Getting rid of branching can therefore aid in safety. Similarly, Verse does not support function calls but one can use the Gallina functions to mimic such calls. These behave like inline functions that are expanded out at the calling site. For the use case of Verse, we believe this is not a limitation as the burden of supporting non-inlined functions rests with the calling high-level language.

We follow a *correct by construction* strategy when it comes to supporting language features like type checking in Verse. As in other EDSLs, programming in Verse involves directly generating the abstract syntax tree, instead of parsing the program from a file. The abstract syntax tree corresponding to a particular Verse language construct is naturally represented as an inhabitant of the associated inductive type. We design such inductive types so that ill-typed Verse programs are not representable. For example, the constructor for the inductive type that captures array indexing requires, as arguments, the array variable, the index, and a proof that the index is within bounds. As a result, out of bound array indexing is not representable.

The main advantage of such a strategy is that we get the type checking in Verse *for free* without writing a type checker and spending time proving its correctness. Here, again, the simplicity of Verse is crucial; for otherwise, the definition of the associated ASTs would have been too complicated to handle.

Following the above strategy, Verse ensures the following type safety (Section 3.3):

- Arithmetic and bitwise operations are only allowed when all the operands are of the same type.
- Targets of assignment or update operations are lvalues.
- Array indexing is always within bounds.

For a user of the Verse EDSL, directly using the inductive type constructors is tedious due to their unnatural syntax. We make use of Coq's notation system to provide necessary syntactic sugar. We also shield the user from proof obligations that arise during array indexing by providing automatic tactics to dispose of them. As a result, the surface syntax of Verse is very close to a high-level language as illustrated in Section 2.

The rest of this section is a brief tour of the internals of Verse illustrating the use of the correct by construction strategy. We concentrate on three inductive types that form the core of Verse — the inductive type `type` of Verse types, the type `VariableT` of program variables, and the inductive type `instruction` of generic Verse instructions.

[3]https://github.com/piyush-kurur/verse-coq/blob/master/src/Verse/Artifact/ChaCha20.v

3.1 Types and kinds

The type system of Verse comprises of the base types, like Word8, Word16, Word32 and Word64, multiwords (SIMD vector types) and arrays. Of these, only words and multiword types can potentially be stored in machine registers. Array elements, on the other hand, need to be addressed indirectly. In Verse, we distinguish these types using a kind system, defined as the inductive type:

```
Inductive kind : Set := direct | memory
```

Types in Verse are defined as type families over `kind`. Following, the correct by construction strategy, the `word` and `multiword` constructors generate only `type direct` whereas the `Array` constructor generates `type memory`.

```
word      :  nat -> type direct
multiword :  nat -> nat -> type direct
Array     :  nat
          -> endian
          -> type direct
          -> type memory
```

Verse only allows arrays of word and multiword types and this is enforced by restricting the type parameter for the `Array` constructor to `type direct`. The `nat` parameter is the array bound. In addition, since arrays are stored in memory, arrays also keep track of endianness. The endianness becomes relevant when generating code for indexing array elements.

3.2 Program variables

One of the important goals of Verse is to provide ways to write generic assembly program fragments which can be ported and reused across architectures. Arithmetic and bitwise instructions which are the most relevant instructions for programming cryptographic primitives are supported across architectures. However, their register sets are often very different. We solve this problem by making instructions parametric over the register set. In generic Verse programs, variables are typed by an element of the type universe `VariableT`.

```
VariableT = forall k : kind, type k -> Type
```

Any `var` : `VariableT` is a type and elements `A B . . .` : `var k ty` are the actual program variables of type `ty`. Generic Verse programs are therefore parameterised over `VariableT`. During code generation, this generic variable type is instantiated by the type of machine registers which is also part of the universe `VariableT`.

3.3 Operands and Instructions

Instructions in Verse are defined as an inductive type parameterised over the universe `VariableT` of variable types.

```
instruction : VariableT -> Type

code = fun v : VariableT => list (instruction v)
         : VariableT -> Type
```

A code fragment over the variable type `v` is just a `list` (`instruction v`). Recall that the first stage of programming in Verse is to write a generic program. In Verse, generic programs that use `n` variables of types $\mathtt{ty}_1$, ..., $\mathtt{ty}_n$ are functions of the type:

$$\forall(\mathtt{var} : \mathtt{VariableT}),\ \mathtt{var}\ \mathtt{ty}_1 \rightarrow \ldots \rightarrow \mathtt{var}\ \mathtt{ty}_n \rightarrow \mathtt{code}\ \mathtt{var}.$$

The constructors of the `instruction` type expect operands captured by the `arg` inductive type. We now look at the inductive types `instruction` and `arg` in some detail and explain how we use the correct by construction strategy to achieve type checking of Verse programs.

Operands to instructions can either be constants or program variables, or an index into an array held inside a program variable. This means that arguments are themselves parameterised by program variables. We also want to ensure that a constant is not used as the target of an assignment. This we achieve using the `argKind` type.

```
Inductive argKind := lval | rval

arg : VariableT -> argKind
                -> forall k : kind, type k
                -> Type
```

Consider an instruction like `x += y`. The constructor associated with this instruction is:

```
update2 : forall (v : VariableT) (ty : type direct),
                 binop ->
                 arg v lval direct ty ->
                 arg v rval direct ty ->
                 assignment v
```

This constructor, when used to encode the instruction `x += y`, enforces the following:

- The argument `x`, is an lval and hence cannot be a constant operand. This is achieved by making sure that the `const` constructor for `arg` generates an `arg` of `argKind rval`.
- Both `x` and `y` are of the same direct type.

One other constructor of `arg` bears description. The constructor `index`, that constructs an array indexing operand like `A [- i -]` in programs has the following type.

```
index : forall (v : VariableT) (aK : argKind),
        forall {b : nat}
               {e : endian}{ty : type direct}
               (x : v memory (Array b e ty)),
        { i : nat | i < b } -> arg aK direct ty
```

This encodes an array of size `b` being dereferenced at index `i`. However, indices are constrained using the sigma type {`i` : `nat` | `i` < `b`} and out of bounds access is ruled out.

The `instruction` type itself, for the most part, encapsulates assignment and update operations with arithmetic and binary operators. However, the `MOVE` instruction is not as typical. We defer it's description to the next subsection.

3.4 Endian safety

Cryptographic primitives like hashes and ciphers often work on data by chunking them into blocks of fixed size and working one block at a time. The primitives treat these blocks as arrays of a particular word type and perform various transformations on them. If this word type is multiple bytes long, one needs to specify the byte order used to encode these words. For example, the SHA512 hashing algorithm treats its input as arrays of length 16 of Word64's encoded in big endian. Often, implementations that work on one architecture fail miserably on another with a different byte order. In the context of testing and implementing AES candidates [Gladman 1999] reports that incorrect handling of endianness was one of the biggest sources of bugs. Verse has inbuilt support for handling this issue; all one needs to do is to declare the array with the correct endianness. The code generation takes care of the appropriate conversions.

In Verse, array types are parameterised by endianness apart from their length and content type. We also allow the possibility of arrays encoding their elements in *host* endianness to represent data for which endianness either does not matter or is already taken care of at a higher level.

```
endian : Type := bigE | littleE | hostE.
Array     :  nat
          -> endian
          -> type direct
          -> type memory
```

Having distinguished arrays of different endianness at the type level, Verse ensures proper endian conversions while indexing. Consider a multi-byte type, say `Word64`, and program variables `X` and `A` of types `Word64` and `Array` 42 `bigE` `Word64`. On a big endian machine, the instructions `X` ::== `A`[- 10 -] and `A`[- 10 -] ::== `X` will be compiled into loads and stores. However on a little endian machine, the code generator will compile `X` ::== `A`[- 10 -] to a load followed by a byteswap on `X` whereas the assignment `A`[- 10 -] ::== `X` will become a byteswap on `X` to get to big endian encoding, a store of `X` into `A`[- 10 -], and finally a byteswap on `X` to restore the value back.

Verse also supports a more efficient `MOVE X TO A`[- 10 -] which can be used instead of `A`[- 10 -] ::== `X` when it is known that the value of `X` is no longer required. The semantics of Verse for the `MOVE` instruction make the contents of `X` invalid for subsequent use, effectively allowing the generated code to omit the final byte swap and hence being slightly more efficient. In a sense, the `MOVE` instruction has the semantics of ownership transfer as in Rust. Programs with `MOVE` instructions need to be checked for *move violations*, i.e. an invalidated variable should not be used subsequently without being reassigned first. We formalise a generic semantics for the Verse language in Section 4 where errors which involve the use of invalidated variables can be caught.

Apart from reducing endian conversion errors, the automatic handling of endianness helps in portability as well. Consider two different architectures both of which have comparable instruction set but which differ in their endianness. It would have been a pity if a generic routine needed to be rewritten just to take care of endianness. Using verse we can write a single function and then make it work on both the architecture.

3.5 Code generation

Before looking at the internals of code generation, we look at how architectures are specified. An architecture is modularly built out of the following:

- A `machineVar` : `VariableT` which corresponds to register and stack variables in the architecture.
- Frame management routines to handle the calling convention.
- The actual translation of supported instructions to assembly code.

Consider a generic program i.e. a Gallina function

`prog` : $\forall$(`var` : `VariableT`), `var ty`$_1$ $\rightarrow \ldots \rightarrow$ `var ty`$_n$ $\rightarrow$ `code var`.

The architecture module is equipped to generate code from `prog′` := `prog machineVar` with some help from the user. It needs a separation of the variables into parameters and locals, and explicit register allocations for a subset of the local variables (the rest are spilled onto the stack by the frame management routine). Provided with this information the code generation proceeds as follows:

- Parameters are allocated according to the calling convention of the architecture the code is being compiled down to.
- The local variables are allocated onto registers or the stack according to specification.
- The fully instantiated generic program is now translated to assembly code.

Recall that an architecture need not support all the instructions or types that Verse provides. The above process can, therefore, throw errors of `UnsupportedType`, `UnsupportedInstruction`, as also `UnavailableRegister` for when a local allocation on a register conflicts with the calling convention.

Typically, assembly language fragments involve a lot of program variables, and defining functions like `prog` directly is tedious. The sectioning mechanism of Coq provides a convenient way of defining such functions and also packaging meta information like the parameter, stack, and register lists conveniently. As demonstrated in Section 2, these Coq features together with the helper tactics and functions provided by Verse provide a coding environment that hides most of these internals.

4 SEMANTICS

Following the strategy of [Chlipala 2013], we define an interpreter for generic Verse programs in Coq. We first map types in Verse to types in Coq. The simplest types in Verse are the word types (`Word8`, `Word16`, . . .). A natural way to interpret them is to use bitvectors. Besides the word types, Verse supports multi-words and arrays. Multi-words correspond to vectors of words and support pointwise operations on their coordinates. Therefore, multiwords are interpreted as vectors of their base word type. We capture this meaning by a function.

`Definition` `typeDenote` : $\forall$ {`k` : `kind`}, `type k` $\rightarrow$ **Type**.

Apart from some special instructions like `MOVE`, instructions in Verse look like `X = Y op Z` for some operator `op`. To interpret these instructions we need a *State* which maps variables to optional values. States map uninitialised or clobbered variables to `None`.

`Definition` `State` := $\forall$ `k` (`ty` : `type k`), `var ty` $\rightarrow$ `option` (`typeDenote ty`).

Every instruction denotes a map from state to state.

`Definition` `instructionDenote` : `instruction var` $\rightarrow$ `State` $\rightarrow$ `State` + {`EnvError`}.

Finally, code semantics is just a lift of `instructionDenote` to `list` (`instruction var`).

4.1 Generalised semantics

The above interpretation is what we call the standard semantics for Verse programs. We can generalise this to a much more abstract notion. Let us explore this idea by distilling out the essence of the semantics.

- We define what the types of Verse mean via `typeDenote`.
- We then give a meaning to instructions via `instructionDenote`.

Notice that the function `typeDenote` is completely specified by giving an interpretation for the word types. In other words, given the function `wordDenote : ∀ n : nat → Type`, we can lift it to an appropriate `typeDenote`. We do not need to stick to the standard semantics and can choose `wordDenote` to be an arbitrary type family on `nat`.

The definition of `instructionDenote` is also fully specified by an interpretation of the operators, i.e. given a function `opDenote : ∀ n, operator → wordDenote n → wordDenote n → wordDenote n` (ignoring variance in arity for simplicity), we can lift it to an `instructionDenote`. We can thus build a semantics for Verse from any `wordDenote` and an `opDenote` over it.

Even the simplest case of such generalised semantics, where `wordDenote` is the constant function `fun _ ⇒ unit` and `opDenote op` is the function `fun _ _ ⇒ tt`, has a non-trivial application. This semantics propagates validity instead of values of the underlying variables, thus providing a means to check for invalid variable use. Direct applications are detection of move violation or use of uninitialised variables.

4.2 Semantics for bound checking

Cryptographic implementations often require arithmetic operations over *big words*, i.e. words with size larger than the word size of the machine. Such a big word is split across multiple machine registers. Given two such words, we need routines to multiply and add them. A prerequisite to correctness of such routines is the absence of overflows in the 64-bit arithmetic that the machine actually performs. For concreteness, consider an implementation of the Poly1305 message authentication algorithm on a 64-bit machine. It involves modular arithmetic over the prime modulus $p = 2^{130} - 5$. Elements $a \mod p$ can be represented as 130-bit words. A common representation of these 130-bit words is as five 26-bit words $a_0, \ldots, a_4$ stored in 64-bit registers. This seemingly wasteful representation is carefully tailored to allow the multiplication routine to go through on a 64-bit machine.

Consider a custom semantics that keeps track of the *upper* and *lower bounds* instead of the actual values of variables, i.e. `wordDenote n` is the type `nat × nat`, and `opDenote op` computes bounds for the operational result from the bounds on its arguments. It is clear that this semantics can, in particular, be used to check overflow errors.

Using bound semantics to check for overflow errors should target the actual instructions that perform the arithmetic and not the entire program. The tendency to refactor code fragments heavily in Verse makes this style of verification natural. Coding within this style we never found reasons to execute the bounded semantics on loops which simplifies verification.

5 CHALLENGES AND FUTURE WORK

A drawback of the style of semantics that we discussed above is that they are slow to execute; others have reported similar slow down [Kennedy et al. 2013]. This was one of the primary reasons for our move towards generic semantics. Empty semantics and bound semantics are efficient enough for the applications we described above. Nonetheless we would like to improve the performance of standard semantics towards writing functional equivalence proofs.

Recall that Verse code is meant to be called via FFI from a high level language like OCaml or Haskell. Writing tightly specified primitives is moot without a facility to transfer restrictions on the parameters to the calling function. Coq's ability to extract code in these calling languages means that it can act as a bridge between the two worlds. In the future, we plan to implement a cryptographic library which uses Coq to implement both the high-level features (via extraction) as well as the low-level primitives (via Verse).

On the code generating front, Verse currently just pretty prints the AST to the appropriate assembly language instructions. What this means is that Verse's code generation detracts from it being an end-to-end verified compiler. Projects like CompCert [Leroy et al. 2012] on the other hand have a processor model, i.e. an inductive type for assembly instructions and a semantics in Coq for the instructions, to prove correctness of their code generation phases. However, this is not to be considered as a serious limitation as Verse instructions translate more or less one to one to machine instructions with no serious code transformations in between. Even if routed through a processor model, there would be a stage where the assembly language instructions, typically represented by inductive types, would need to be pretty printed. Nonetheless, there are some advantages of using a processor model. In particular, if one is able to integrate the processor models of a well established project like CompCert, we could gain additional trust in the process, not to mention the additional benefit of being able to target multiple targets (Verse currently only targets X86_64 and C).

There are other targets that are equally interesting. We could target languages like Dafny [Leino 2010]much like the way we targeted C. Finally, even our C backend could gain additional trust by generating C code annotated with specifications that can be automatically checked by systems like Frama-C [Cuoq et al. 2012]. We hope to pursue some of these ideas in future works.

6 RELATED WORK

The primary motivation for Verse has been the qhasm project [Barbosa et al. 2011; Bernstein 2007] which probably was the first attempt to provide a low-level language specifically to program cryptographic primitives. Qhasm reduces the burden of targeting multiple architectures by providing C-like notation for instructions that are common across architectures. However, unlike Verse, qhasm is a set of string replacements and hence provides none of the high-level features, for refactoring repetitive coding patterns. One often needs a macro processor [Käsper and Schwabe 2009; Schwabe 2015] to make up for these limitations. Finally, qhasm does not address issues of type safety.

One could also approach the above problem by designing a full programming language targeted specifically for cryptographic primitives, i.e. a DSL instead of an EDSL. The CAO programming language [Barbosa et al. 2011] was an early attempt. A more recent and well maintained project is Cryptol [Pike et al. 2006] where one can provide very high-level functional specifications to cryptographic primitives. Cryptol, however, is designed to target dedicated hardware implementations as opposed to software implementations.

A very recent example of a DSL with scope much closer to Verse is the Jasmin programming language [Almeida et al. 2017]. Unlike Verse, it is a full-fledged programming language together with a certified compiler written in Coq. Basic type safety is guaranteed by the Jasmin compiler. Programs are then functionally embedded into Dafny which checks for errors like out of bound array indexing. In addition Jasmin programs can be annotated with user specifications that get translated to Dafny annotations by the compiler. Dafny, in turn, checks correctness by using the SMT solver Z3.
Our approach is definitely much more light-weight

- We do not have a parsing stage (Jasmin does not verify its parser [Almeida et al. 2017, Section 5.3]) as we generate the AST directly. Validating parsers is non-trivial and can be a formalisation burden in an end-to-end certified compiler [Jourdan et al. 2012].
- Features like word level polymorphism is merely an idiom for us, whereas it would be a major language feature for a DSL (together with its type checking algorithms and their correctness proofs).
- In addition, our focus on the bare essentials required for implementing cryptographic primitives meant that we could follow a correct by construction strategy even for things like array bound checking.

Jasmin does support some high-level language features like genuine loops as opposed to unrolled ones and conditional branches. However, for speed and side-channel resistance, it is often the case that real world implementations unroll loops and avoid conditional branches. By focusing only on the inner loop of a cryptographic primitive, Verse does not significantly compromise on features. In fact, endian correctness is a feature that adds both to the correctness and portability of cryptographic implementations, which, to the best of our knowledge, is not supported by Jasmin.

The main objective of Verse is to be a vehicle for writing cryptographic primitives, as much as possible, in a portable way. The project Vale [Bond et al. 2017] addresses a different use case where the goal is to add safety to an already existing code base of machine-specific assembly language, for example, the code base of OpenSSL. This they achieve by embedding an annotated version of the assembly language into Dafny. The annotations carry safety conditions that are then handled by a SAT solver.

Cryptographic primitives are not an end on their own and often are just cogs in the larger scheme of secure cryptographic libraries and applications. Another EDSL with a similar outlook is Low* [Protzenko et al. 2017]. It is embedded into F* and, barring specifications, Low* code is close to the C code it compiles down to. We look to take it a step further and provide a usable interface to target assembly code generation directly. Also, being embedded in Coq means that Verse can work well with other projects in Coq with related goals.

- Verse can be the targets for other higher level compilers or other efforts in Coq like [Erbsen et al. 2018] where they develop efficient implementations by successive refinement. In such cases, code is often generated by Coq functions and the abstract syntax tree as an inductive type can help.
- As mentioned in the future works (Section 5), Verse can play an important role in an end to end verified cryptographic library in Haskell where the higher-level function (in Haskell) is extracted from its Coq implementation and low-level primitives are encoded in Verse.

REFERENCES

J B Almeida, M Barbosa, G Barthe, A Blot, B Grégoire, V Laporte, T Oliveira, H Pacheco, and B Schmidt. 2017. Jasmin: High-Assurance and High-Speed Cryptography. , 1807–1823 pages. publications/ccs17.pdf

Manuel Barbosa, Andrew Moss, and Peter Schwabe. 2011. *CAO and qhasm compiler tools*. Technical Report. Computer Aided Cryptography Engineering. http://www.cspforum.eu/32_CACE_D1.3_CAO_and_qhasm_compiler_tools_Jan11.pdf

Daniel J Bernstein. 2007. Writing high speed software. https://cr.yp.to/qhasm.html

Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 917–934. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond

Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press.

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-c. In *International Conference on Software Engineering and Formal Methods*. Springer, 233–247.

Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2018. Systematic Generation of Fast Elliptic Curve Cryptography Implementations. https://people.csail.mit.edu/jgross/personal-website/papers/2018-fiat-crypto-pldi-draft.pdf

Brian Gladman. 1999. Implementation experience with AES candidate algorithms. In *Proc. of Second AES Candidate Conference (AES2), March 1999.*

Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) Parsers. In *Proceedings of the 21st European Conference on Programming Languages and Systems (ESOP'12)*. Springer-Verlag, Berlin, Heidelberg, 397–416. https://doi.org/10.1007/978-3-642-28869-2_20

Emilia Käsper and Peter Schwabe. 2009. Faster and Timing-Attack Resistant AES-GCM. In *Cryptographic Hardware and Embedded Systems - CHES 2009*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–17.

Andrew Kennedy, Nick Benton, Jonas B. Jensen, and Pierre-Evariste Dagand. 2013. Coq: The World's Best Macro Assembler?. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming (PPDP '13)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/2505879.2505897

K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 348–370.

Xavier Leroy et al. 2012. The CompCert verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt* (2012).

Lee Pike, Mark Shields, and John Matthews. 2006. A Verifying Core for a Cryptographic Language Compiler. In *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 '06)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/1217975.1217977

Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, et al. 2017. Verified low-level programming embedded in F. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 17.

Peter Schwabe. 2015. maq - a preprocessor for qhasm. https://cryptojedi.org/programming/maq.shtml.